Migrating Win32 C/C++ applications to Linux on POWER, Part 1: Process, thread, and shared memory services

Nam Keung
Chakarat Skawratananond

June 10, 2004

This article covers Win32 API mapping, particularly process, thread, and shared memory services to Linux on POWER. The article can help you decide which of the mapping services best fits your needs. The author takes you through the APIs mapping he faced while porting a Win32 C/C++ application.

Overview

There are many ways to port and migrate from the Win32 C/C++ applications to the pSeries platform. You can use freeware or 3rd party tools to port the Win32 application code to move to Linux. In our scenario, we decided to use a portability layer to abstract the system APIs call. A portability layer will offer our application the following benefits:

- Independence from the hardware
  - Independence from the operating system
  - Independence from changes introduced from release to release on operating systems
- Independence from operating system API styles and error codes
- Ability to uniformly place performance and RAS hooks on calls to the OS

Because the Windows environment is quite different from the pSeries Linux environment, porting across UNIX platforms is considerably easier than porting from the Win32 platform to the UNIX platform. This is expected, as many UNIX systems share a common design philosophy and provide a lot of similarities at the application layer. However, the Win32 APIs are in limited in the task of porting to Linux. This article identifies issues due to differences in design between Linux and Win32.

Initialization and termination

On Win2K/NT, the initialization and termination entry point for a DLL is the _DLL_InitTerm function. When each new process gains access to the DLL, this function initializes the necessary environment for the DLL. When each new process frees its access to the DLL, this function terminates the DLL for that environment. This function is called automatically when you link to

the DLL. For applications, an additional initialization and termination routine is included in the _DLL_InitTerm function.

On Linux, the GCC has an extension that allows specifying that a function should be called when the executable or shared object containing it is started up or finished. The syntax is `__attribute__((constructor))` or `__attribute__((destructor))`. These are basically the same as constructors and destructors to replace the _init and _fini functions from the glibc library.

The C prototypes for these functions are:

```
void __attribute__ ((constructor)) app_init(void);
void __attribute__ ((destructor)) app_fini(void);
```

```
Win32 sample
_DLL_InitTerm(HMODULE modhandle, DWORD fdwReason, LPVOID lpvReserved)
{
    WSADATA   Data;
    switch (fdwReason)
    {
        case DLL_PROCESS_ATTACH:
        if (_CRT_init() == -1)
           return 0L;
          /* start with initialization code */
          app_init();
          break;
    case DLL_PROCESS_DETACH:
         /* Start with termination code*/
         app_fini();
         _CRT_term();
         break;
    â#¦..
    default:
          /* unexpected flag value - return error indication */
            return 0UL;
    } return 1UL;  /* success */
}
```

Process service

The Win32 process model has no direct equivalents to `fork()` and `exec()`. Rather than always inherit everything as is done in Linux with the `fork()` call, the `CreateProcess()` accepts explicit arguments that control aspects of the process creation, such as file handle inheritance.

The CreateProcess API creates a new process containing one or more threads that run in the context of the process, and there is no relationship between the child and parent processes. On Windows NT/2000/XP, the returned process ID is the Win32 process ID. On Windows ME, the returned process ID is the Win32 process ID with the high-order bit stripped off. When a created process terminates, all the data associated with the process is erased from the memory.

To create a new process in Linux, the `fork()` system call duplicates the process. After a new process is created, a parent and child relationship is automatically established and the child process inherits all the attributes from the parent process by default. Linux creates a new process using a single system call with no parameters. The `fork()` returns the child's process ID to the parent and none to the child.

Win32 processes are identified both by handles and process IDs, whereas Linux has no process handles.

Process mapping table

Win32	Linux
CreateProcess	fork() execv()
TerminateProcess	kill
ExitProcess()	exit()
GetCommandLine	argv[]
GetCurrentProcessId	getpid
KillTimer	alarm(0)
SetEnvironmentVariable	putenv
GetEnvironmentVariable	getenv
GetExitCodeProcess	waitpid

Create process service

In Win32, the first argument of `CreateProcess()` specifies the program to run, and the second argument provides the command line arguments. CreateProcess takes other process parameters as arguments. The second-to-last argument is a pointer to a STARTUPINFORMATION structure, which specifies the standard devices for the process and other start up information about the process environment. You need to set the hStdin, hStdout, and hStderr members of STARTUPINFORMATION structure before passing its address to CreateProcess to redirect standard input, standard output, and standard error of the process. The last argument is a pointer to a PROCESSINFORMATION structure, which is filled up by created processes. Once the process starts, it will contain, among other things, the handle to the created process.

```
Win32 example
PROCESS_INFORMATION    procInfo;
STARTUPINFO         startupInfo;
typedef DWORD          processId;
char                   *exec_path_name
char                   *_cmd_line;

GetStartupInfo( &startupInfo );     // You must fill in this structure
if( CreateProcess( exec_path_name,  // specify the executable program
                       _cmd_line,   // the command line arguments
                       NULL,       // ignored in Linux
                       NULL,       // ignored in Linux
                       TRUE,       // ignored in Linux
                       DETACHED_PROCESS | HIGH_PRIORITY_CLASS,
                       NULL,       // ignored in Linux
                       NULL,       // ignored in Linux
               &startupInfo,
                    &procInfo))
        *processId = procInfo.dwProcessId;
else
{
        *processId = 0;
          return RC_PROCESS_NOT_CREATED;
}
```

In Linux, the processId is an integer. The search directories in Linux are determined by the PATH environment variable (exec_path_name).The `fork()` function makes a copy of the parent, including the parent's data space, heap, and stack. The `execv()` subroutine uses the exec_path_name to pass the calling process current environment to the new process.

This function replaces the current process image with a new process image specified by exec_path_name. The new image is constructed from a regular, executable file specified by exec_path_name. No return is made because the calling process image is replaced by the new process image.

```
Equivalent Linux code
#include <stdlib.h>
#include <stdio.h>

int    processId;
char  *exec_path_name;
char  * cmd_line ;

cmd_line = (char *) malloc(strlen(_cmd_line ) + 1 );

if(cmd_line == NULL)
         return RC_NOT_ENOUGH_MEMORY;

strcpy(cmd_line, _cmd_line);

if( ( *processId = fork() ) == 0 )  // Create child
 {
         char  *pArg, *pPtr;
         char  *argv[WR_MAX_ARG + 1];
         int   argc;
         if( ( pArg = strrchr( exec_path_name, '/' ) ) != NULL )
                pArg++;
         else
                pArg = exec_path_name;
         argv[0] = pArg;
         argc = 1;

         if( cmd_line != NULL && *cmd_line != '\0' )
         {

               pArg = strtok_r(cmd_line, " ", &pPtr);

               while( pArg != NULL )
               {
                              argv[argc] = pArg;
                              argc++;
                              if( argc >= WR_MAX_ARG )
                              break;
                              pArg = strtok_r(NULL, " ", &pPtr);
                }
         }
         argv[argc] = NULL;

         execv(exec_path_name, argv);
         free(cmd_line);
         exit( -1 );
}
else if( *processId == -1 )
{
           *processId = 0;
           free(cmd_line);
           return RC_PROCESS_NOT_CREATED;
}
```

Terminate process service

In the Win32 process, the parent and child processes may require different access to an object identified by a handle that the child inherits. The parent process can create a duplicate handle with the desired access and inheritability. The Win32 sample code uses the following scenario to terminate a process:

- Use OpenProcess to get the handle of the specified process
- Use GetCurrentProcess for its own handle
- Use DuplicateHandle to obtain the handle from the same object as the original handle

If the function succeeds, use the TerminateThread function to release the primary thread on the same process. The TerminateProcess function is then used to unconditionally cause a process to exit. It initiates termination and returns immediately.

```
Win32 sample code
if( thread != (HANDLE) NULL )
{
    HANDLE thread_dup;
    if( DuplicateHandle( OpenProcess(PROCESS_ALL_ACCESS, TRUE,  processId),
                                    thread,
                                    GetCurrentProcess(),
                                    &thread_dup,  //Output
                                    0,
                                    FALSE,
                                    DUPLICATE_SAME_ACCESS ))
      {
            TerminateThread( thread_dup, 0);
       }
}
TerminateProcess(OpenProcess(PROCESS_ALL_ACCESS, TRUE, processId),
   (UINT)0 );
```

In Linux, use the kill subroutine to send the SIGTERM signal to terminate the specified process (processId). Then call the waitpid subroutine with the WNOHANG bit set. This checks the specified process and is terminated.

```
Equivalent  Linux code
pid_t nRet;
int status;

kill( processId, SIGTERM );
nRet = waitpid( processId, &status, WNOHANG);  //Check specified
   process is terminated
```

Process still exists service

The Win32 OpenProcess returns the handle to the specified process (processId). If the function succeeds, the GetExitCodeProcess will retrieve the status of the specified process and checks whether the process status is STILL_ACTIVE.

```
Win 32 sample
HANDLE      nProc;
DWORD       dwExitCode;

nProc = OpenProcess(PROCESS_ALL_ACCESS, TRUE, processId);
if ( nProc != NULL)
{
     GetExitCodeProcess( nProc, &dwExitCode );
     if (dwExitCode == STILL_ACTIVE )
            return RC_PROCESS_EXIST;
     else
            return RC_PROCESS_NOT_EXIST;
}
else
     return RC_PROCESS_NOT_EXIST;
```

In Linux, use the kill subroutine to send the signal specified by the `Signal` parameter to the process specified by the `Process` parameter (processId). The Signal parameter is a null value, the error checking is performed, but no signal is sent.

```
Equivalent Linux code
if ( kill ( processId, 0 ) == -1 && errno == ESRCH ) // No process can
                                      be found
                              // corresponding to processId
         return RC_PROCESS_NOT_EXIST;
else
        return RC_PROCESS_EXIST;
```

Thread model

A *thread* is a basic entity to which the system allocates CPU time; each thread maintains information to save its "context" while waiting to be scheduled. Each thread can execute any part of the program code and share global variables of the process.

LinuxThreads is a pthreads compatible thread system built on top of the `clone()` system call. Because threads are scheduled by the kernel, LinuxThreads supports blocking I/O operations and multiprocessors. However, each thread is actually a Linux process, so the number of threads a program can have is limited to the total number of processes allowed by the kernel. The Linux kernel does not provide system calls for thread synchronization. The Linux Threads library provides additional code to support operations on mutex and condition variables (using pipes to block threads).

For signal handling when coupled with the LinuxThreads, each thread inherits a signal handler if one was registered by a parent process that spawned the thread. Only the new features supported in Linux Kernel 2.6 and higher will include the improved POSIX threading support, such as the Native POSIX Thread Library for Linux (NPTL).

Important parts of the thread model are the thread synchronization, wait functions, thread local storage, and initialization and termination abstraction. Below, the thread services are only addressed as:

- A new thread is created, and the threadId is returned
- The current new thread can be terminated by invoking the pthread_exit function

Thread mapping table

Win32	Linux
_beginthread	pthread_attr_init pthread_attr_setstacksize pthread_create
_endthread	pthread_exit
TerminateThread	pthread_cancel
GetCurrentThreadId	pthread_self

Thread Creation

The Win32 application uses the C runtime libraries instead of using the Create_Thread APIs. The routines of _beginthread and _endthread are used. These routines take care of any reentrancy and memory leak problems, thread local storage, initialization, and termination abstraction.

Linux uses the pthread library call `pthread_create()` to spawn a thread.

The threadId is returned as an output parameter. A set of parameters are passed for creating a new thread. The arguments execute a function when a new thread is created. The stacksize argument is used as the size, in bytes, of the new thread's stack, and the actual argument is to be passed to the function when the new thread starts executing.

Specify the thread procedure (function)

The creating thread must specify the starting function of the code that the new thread is to execute. The starting address is the name of the threadproc function with a single argument, threadparam. If the call succeeds in creating a new thread, the threadId is returned. The typedef for Win32 threadId is a HANDLE. The typedef for Linux threadId is pthread_t.

threadproc
The thread procedure (function) to be executed. It receives a single void parameter.

threadparam
The parameter to be passed to the thread when it begins execution.

Set the stack size

In Win32, the thread stack is allocated automatically in the memory space of the process. The system increases the stack as needed and frees it when the thread terminates. In Linux, the stack size is set in the pthread attributes object; the pthread_attr_t is passed to the library call `pthread_create()`.

```
Win32 sample
int  hThrd;
DWORD          dwIDThread;
unsigned       stacksize;
void           *thrdparam; //parameter to be passed to the thread when it
                                 //begins execution
HANDLE         *threadId;

if( stacksize < 8192 )
     stacksize = 8192;
else
     stacksize = (stacksize/4096+1)*4096;
```

```
     hThrd = _beginthread( thrdproc,    // Definition of a thread entry
                                                                  //point
                                NULL,
                                stacksize,
                                thrdparam);
if (hThrd == -1)
     return RC_THREAD_NOT_CREATED);
     *threadId = (HANDLE) hThrd;
_____________________________________________________________________
                          Equivalent Linux code

#include <pthread.h>

pthread_t   *threadId;
void         thrdproc  (void *data);  //the thread procedure (function) to
                                            //be executed.
                     //It receives a single void parameter
void        *thrdparam; //parameter to be passed to the thread when it
                              //begins execution
pthread_attr_t  attr;
int             rc = 0;

if (thrdproc == NULL || threadId == NULL)
      return RC_INVALID_PARAM);

if (rc = pthread_attr_init(&attr))
     return RC_THREAD_NOT_CREATED);  // EINVAL, ENOMEM

if (rc = pthread_attr_setstacksize(&attr, stacksize))
     return RC_THREAD_NOT_CREATED);   // EINVAL, ENOSYS

if (rc = pthread_create(threadId, &attr, (void*(*)(void*))thrdproc,
   thrdparam))
     return RC_THREAD_NOT_CREATED);      // EINVAL, EAGAIN
```

In Win32, one thread can terminate another thread with the TerminateThread function. However, the thread's stack and other resources will not be deallocated. This is recommended if the thread terminates itself. In Linux, the pthread_cancel method terminates execution of the thread identified by the specified threadId.

Terminate thread service

Win32	Linux
TerminateThread((HANDLE *) threadId, 0);	**pthread_cancel**(threadId);

Thread state

In Linux, threads are created in joinable state by default. Another thread can be synchronize with the thread's termination and recover its termination code using the function `pthread_join()`. The thread resources of the joinable thread are released only after it is joined.

Win32 uses `WaitForSingleObject()` to wait for a thread to terminate.

Linux uses pthread_join to do the same.

Win32	Linux
unsigned long rc;	unsigned long rc=0;

rc = (unsigned long) **WaitForSingleObject** (threadId, INIFITE);	rc = **pthread_join**(threadId, void **status);

In Win32, `_endthread()` is used to end the execution of the current thread. In Linux, it is recommended to use `pthread_exit()` to exit a thread to avoid implicitly calling the exit routine. In Linux, the retval is the return value of the thread, and it can be retrieved from another thread by calling `pthread_join()`.

End the execution of current thread service

Win32	Linux
_endthread();	**pthread_exit**(0);

In the Win32 process, The GetCurrentThreadId function retrieves the thread identifier of the calling thread. Linux uses the `pthread_self()` function to return the calling thread's ID.

Get the current thread ID service

Win32	Linux
GetCurrentThreadId()	**pthread_self()**

The time period for the Win32 Sleep function is in milliseconds and can even be INFINITE, in which case the thread will never resume. The Linux sleep function is similar to Sleep, but the time periods are measured in seconds. To obtain the millisecond resolution, use the nanosleep function to provide the same service.

Sleep service

Win32	Equivalent Linux code
Sleep (50)	struct timespec timeOut,remains; timeOut.tv_sec = 0; timeOut.tv_nsec = 500000000; /* 50 milliseconds */ **nanosleep**(&timeOut, &remains);

The Win32 SleepEx function suspends the *current thread* until one of the following occurs:

- An I/O completion callback function is called
- An asynchronous procedure call (APC) is queued to the thread
- The minimum time-out interval elapses

Linux uses the sched_yield to do the same thing.

Win32	Linux
SleepEx (0,0)	**sched_yield()**

Shared memory service

Shared memory allows multiple processes to map a portion of their virtual address to a common memory region. Any process can write data to a shared memory region, and the data are readable

and modified by other processes. The shared memory is used to implement an interprocess communication media. However, shared memory does not provide any access control for processes that use it. It is a common practice to use "locks" along with shared memory.

A typical usage scenario is:

1. A server creates a shared memory region and sets up a shared lock object
2. A client can attach the shared memory region created by the server
3. Both the client and server can use the shared lock object to get access to the shared memory region
4. The client and server can query the location of the shared memory resource

Shared memory mapping table

Win32	Linux
CreateFileMaping, OpenFileMapping	mmap shmget
UnmapViewOfFile	munmap shmdt
MapViewOfFile	mmap shmat

Create a shared memory resource

Win32 creates a shared memory resource by shared memory-mapped files. Linux uses the shmget/mmap function to access files by directly incorporating file data into memory. The memory areas are known as the shared memory segments.

Files or data can also be shared among multiple processes or threads. But, this requires synchronization between these processes or threads and its handling is up to the application.

The `CreateFileMapping()` reinitializes the commitment of the shared resource to the process if the resource already exists. The call can fail if there is insufficient memory free to handle the erroneous shared resource. `OpenFileMapping()` indicates the shared resource must already exist; this call is merely requesting access to it.

In Win32, the CreateFileMapping does not allow you to grow the file size, but that's not the case in Linux. In Linux, if the resource already exists, it will be re-initialized. It may be destroyed and recreated. Linux creates the shared memory that is accessed using a name. The `open()` system call determines whether the mapping is readable or writable. The parameters passed to `mmap()` must not conflict with the access requested during `open()`. The `mmap()` needs to supply the size of the file as the number of bytes to map.

For 32-bit kernels, there are 4GB virtual address spaces. The first 1 GB is for device drivers. The last 1 GB is for kernel data structures. The 2GB in the middle can be used for shared memory. Currently, Linux on POWER allows 4GB for the kernel and up to 4GB virtual address space for user applications.

Mapping memory access protection bits

Win32	Linux
PAGE_READONLY	**PROT_READ**
PAGE_READWRITE	**(PROT_READ | PROT_WRITE)**
PAGE_NOACCESS	**PROT_NONE**
PAGE_EXECUTE	**PROT_EXEC**
PAGE_EXECUTE_READ	**(PROT_EXEC |PROT_READ)**
PAGE_EXECUTE_READWRITE	**(PROT_EXEC | PROT_READ | PROT_WRITE)**

To find out the allocation of the Linux shared memory, you can look at shmmax, shmmin and shmall under the /proc/sys/kernel directory.

An example to increase shared memory on Linux is:

```
echo 524288000 > /proc/sys/kernel/shmmax
```

The maximum shared memory is increased to 500 MB.

Below is the Win32 sample code, and the equivalent of the Linux mmap implementation, to create a shared memory resource.

```
Win32 sample code
typedef struct
 {
   // This receives a pointer within the current process at which the
   // shared memory is located.
   // The same shared memory may reside at different addresses in other
   // processes which share it.
       void * location;
       HANDLE hFileMapping;
}mem_shared_struct, *mem_shared, *token;

mem_shared_struct   *token;

if ((*token = (mem_shared) malloc(sizeof(mem_shared_struct))) == NULL)
     return RC_NOT_ENOUGH_MEMORY;

if (newmode == new_shared_create)
   (*token)->hFileMapping = CreateFileMapping((HANDLE) 0xFFFFFFFF, NULL,
                 PAGE_READWRITE,
                 0,
                (DWORD) size,
                 (LPSTR) name);
 else
      (*token)->hFileMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS,
                 FALSE,
                 (LPSTR) name);
if ((*token)->hFileMapping == NULL)
{
      free( *token );
      return RC_SHM_NOT_CREATED );
}

(*token)->location = MapViewOfFile((*token)->hFileMapping,
                                   FILE_MAP_READ | FILE_MAP_WRITE,
                                   0, 0, 0);
```

```
if ((*token)->location == NULL)
{
      CloseHandle((*token)->hFileMapping);
            free(*token);
            return RC_OBJECT_NOT_CREATED;
}
```

Equivalent Linux code

```
typedef struct
{
      void    *location;
      int  nFileDes;
      cs_size  nSize;
      char  *pFileName;
}mem_shared_struct, *mem_shared, token;

mode_t mode=0;
int flag=0;
int i, ch='\0';
char   name_buff[128];

if (newmode == new_shared_create)
                  flag = O_CREAT;
else if (newmode != new_shared_attach)
                  return RC_INVALID_PARAM;

if ((*token = (mem_shared) malloc(sizeof(mem_shared_struct))) == NULL)
               return RC_NOT_ENOUGH_MEMORY;

strcpy(name_buff, "/tmp/" );
strcat(name_buff, name );

if(((*token)->pFileName = malloc(strlen(name_buff)+1)) == NULL )
{
      free(*token);
      return RC_NOT_ENOUGH_MEMORY;
}

mode |= S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP;
flag |= O_RDWR;

if(newmode == new_shared_create)
      remove(name_buff);

if(((*token)->nFileDes = open(name_buff, flag, mode)) < 0)
{
      free((*token)->pFileName);
      free(*token);
      return RC_OBJECT_NOT_CREATED;
}

if(newmode == new_shared_create)
{
      lseek((*token)->nFileDes, size - 1, SEEK_SET);
      write((*token)->nFileDes, &ch, 1);
}
if(lseek((*token)->nFileDes, 0, SEEK_END) < size)
{
         free((*token)->pFileName);
         free(*token);
         return RC_MEMSIZE_ERROR;
}
(*token)->location = mmap( 0, size,
                  PROT_READ | PROT_WRITE,
                  MAP_VARIABLE | MAP_SHARED,
                      (*token)->nFileDes,
```

```
                                    0);

if((int)((*token)->location) == -1)
{
       free((*token)->pFileName);
       free(*token);
       return RC_OBJECT_NOT_CREATED;
}

(*token)->nSize = size;strcpy((*token)->pFileName, name_buff);
```

To destroy a shared memory resource, the munmap subroutine unmaps a mapped file region. The munmap subroutine only unmaps regions created from calls to the mmap subroutine. If an address lies in a region that is unmapped by the munmap subroutine and that region is not subsequently mapped again, any reference to that address will result in delivery of a SIGSEGV signal to the process.

Delete a shared memory resource

Win32	Equivalent Linux code
UnmapViewOfFile(token->location); **CloseHandle**(token->hFileMapping);	**munmap**(token->location, token->nSize); **close**(token->nFileDes); **remove**(token->pFileName); free(token->pFileName);

Conclusion

This article covers the Win32 APIs mapping to Linux on POWER regarding the initialization and termination, process, thread, and shared memory services. This is by no means to have a full coverage of all APIs mapping and readers can only be used this information as a reference to migrate to Win32 C/C++ application to POWER Linux.